Programmation en langage C++

Exemples

par **Claude DELANNOY**
Ingénieur de l'ENSEM (École nationale supérieure d'électricité et de mécanique) de Nancy
Ingénieur informaticien au CNRS (Centre national de la recherche scientifique)

Nous proposons quelques exemples de programmes illustrant la plupart des fonctionnalités de C++ que nous venons de présenter dans l'article [S 8 065].

1. Exemple 1 : ensembles d'entiers

Nous allons **réaliser une classe nommée *set_int* permettant de manipuler des ensembles de nombres entiers**. Nous prévoirons les opérateurs suivants (*e* désigne un élément de type *set_int* et *n* un entier) :

— <<, tel que *e*<<*n* ajoute l'élément *n* à l'ensemble *e* ;

— %, tel que *n* % *e* vale 1 si *n* appartient à *e* et 0 sinon ;

— <<, tel que *flot* << *e* envoie le contenu de l'ensemble *e* sur le *flot* indiqué, sous la forme :

[entier_1, entier_2, ... entier_n]

La fonction membre *cardinal* fournira le nombre d'éléments de l'ensemble.

Pour ne pas alourdir l'exemple, le nombre maximal d'entiers que pourra contenir l'ensemble sera précisé au constructeur qui allouera dynamiquement l'espace nécessaire.

1.1 Première ébauche de solution

Naturellement, notre classe comportera, en membres données, le **nombre maximal (*nmax*) d'éléments de l'ensemble**, le **nombre courant d'éléments (*nelem*)** et un **pointeur sur l'emplacement contenant les valeurs de l'ensemble**.

L'**opérateur <<** devra être surdéfini de manière à disposer d'un premier opérande de type *set_int* et d'un second opérande entier. Comme le premier opérande est du type de la classe, nous pouvons le définir sous forme d'une fonction membre (de nom *operator<<*).

En revanche, l'**opérateur %** doit être surdéfini obligatoirement sous la forme d'une fonction amie, puisque son premier opérande n'est pas de type classe. L'opérateur de sortie dans un flot doit, lui aussi, être surdéfini sous la forme d'une fonction amie, mais pour une raison différente : son premier argument est d'un type classe (*ostream*) déjà défini et que l'on ne peut pas modifier.

Voici une première ébauche de ce que pourrait être la déclaration de notre classe *set_int* :

```
#include <iostream>
using namespace std ;
class set_int
{    int * adval ;      // adresse du tableau des valeurs
     int nmax ;         // nombre maxi d   l  ments
     int nelem ;        // nombre courant d   l  ments
   public :
     set_int (int = 20) ;                    // constructeur
     ~set_int () ;                           // destructeur
     int cardinal () ;                       // cardinal de l ensemble
     set_int & operator << (int) ;           // ajout d un   l  ment
     friend int operator % (int, set_int &) ;
                                             // appartenance d un   l  ment
        // envoi ensemble dans un ot
     friend ostream & operator << (ostream &, set_int &) ;
} ;
```

1.2 Surdéfinition du constructeur par recopie

Lorsqu'un argument de type *set_int* est transmis en argument à une fonction, il y appel du constructeur de recopie par défaut. Ce dernier recopie les différents champs de l'objet dans un objet local à la fonction, y compris lorsque ceux-ci sont des pointeurs. Toutefois, dans ce dernier cas, l'emplacement pointé n'est nullement recopié.

Dans ces conditions, on se trouve en présence d'un même emplacement dynamique désigné par deux objets différents. Lors de la destruction de ces deux objets (quel qu'en soit l'ordre), on se trouvera libérer deux fois le même emplacement. Les conséquences dépendent de l'implémentation et peuvent être catastrophiques.

Pour régler ce problème, la solution la plus raisonnable consiste à **définir son propre constructeur de recopie** en faisant en sorte qu'il recopie la partie dynamique de l'objet dans un emplacement qu'il créera. Rappelons que son en-tête doit être de la forme :

```
set_int (set_int & e)
```

Remarques

1 – Les mêmes problèmes se posent pour une fonction renvoyant une valeur de type *set_int*. Ils se trouvent réglés conjointement aux précédents par la définition du constructeur par recopie.

2 – Aucun problème ne se pose pour les arguments de type *set_int* transmis par référence puisqu'alors la fonction appelée travaille directement sur l'objet correspondant à l'argument effectif de l'appel.

1.3 Surdéfinition de l'affectation

L'affectation pose des problèmes voisins des précédents. Considérons deux objets *e1* et *e2* de type *set_int* et une simple instruction telle que :

```
e1 = e2 ;
```

Après son exécution, les deux champs *adval* des deux objets pointeront sur la même partie dynamique. Or, lorsque les destructeurs des objets *e1* et *e2* seront appelés (quel qu'en soit l'ordre), on se trouvera libérer, là aussi, deux fois le même emplacement. Quant à l'ancien emplacement associé à *e2*, il se peut qu'il ne soit plus utilisé ; néanmoins, il ne sera jamais libéré.

Pour régler ces problèmes, la solution la plus raisonnable consiste à **surdéfinir convenablement l'opérateur d'affectation de façon compatible avec la démarche utilisée pour le constructeur de recopie**. Ici, donc, nous ferons en sorte que chaque objet de type *set_int* dispose de sa propre partie dynamique.

On notera que si le problème est voisin de celui de la construction par recopie, il n'en est pas pour autant identique. Quelques différences apparaissent :

— on peut se trouver en présence d'une affectation d'un objet à lui-même ;

— avant affectation, il existe deux objets « complets » (c'est-à-dire avec leur partie dynamique). Dans le cas de la construction par recopie, il n'existait qu'un seul emplacement dynamique, le second étant à créer. On va donc se retrouver ici avec l'ancien emplacement dynamique de *b*. Or, s'il n'est plus référencé par *b*, est-on sûr qu'il n'est pas référencé par ailleurs ?

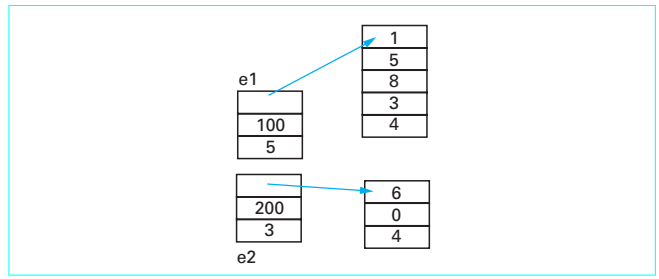
Voici donc comment nous pourrions traiter une affectation telle que *e1* = *e2*, lorsque *e1* est différent de *e2* :

— libération de l'emplacement pointé par *e2* ;

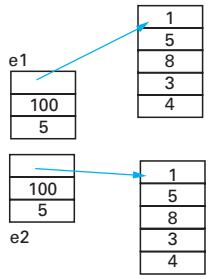
— création dynamique d'un nouvel emplacement dans lequel on recopie les valeurs de l'emplacement pointé par *e1* ;

— mise en place des valeurs des membres données de *e2*.

Supposons qu'avant affectation, la situation se présente ainsi :



Après l'affectation, on aboutit à :



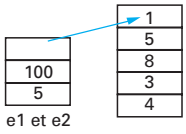
Il reste à régler le cas où *e1* et *e2* correspondent au même objet.

Or, la surdéfinition de l'opérateur d'affectation peut se faire indifféremment avec l'un des deux en-têtes suivants :

```
operator = (set_int & e)
operator = (set_int  e)
```

Si la transmission de *e1* à l'opérateur d'affectation a lieu par valeur et si le constructeur par recopie a été redéfini de façon appropriée (par création d'un nouvel emplacement dynamique), l'algorithme proposé fonctionnera sans problème.

En revanche, si la transmission de *e1* a lieu par référence, on abordera l'algorithme avec cette situation :



L'emplacement dynamique associé à *b* (donc aussi à *a*) sera libéré avant que l'on tente de l'utiliser pour le recopier dans un nouvel emplacement. La situation sera alors catastrophique.

Enfin, il faut décider de la valeur de retour fournie par l'opérateur. À ce niveau, tout dépend de l'usage que nous souhaitons en faire :

— si nous nous contentons d'affectations simples (*e2* = *e1*), nous n'avons besoin d'aucune valeur de retour (*void*) ;

— en revanche, si nous souhaitons pouvoir traiter une affectation multiple ou, plus généralement, faire en sorte que (comme on peut s'y attendre !) l'expression *e2* = *e1* ait une valeur (probablement celle de *e2* !), il est nécessaire que l'opérateur fournisse une valeur de retour.

Nous choisirons ici la seconde possibilité qui a le mérite d'être plus générale.

Voici la **déclaration définitive** de notre classe *set_int* (supposée placée ici dans un fichier nommé *setint.h*) :

Déclaration de la classe *set_in*

```
        /*  fichier SETINT.H  : déclaration de la classe set_int */
#include <iostream>
using namespace std ;
class set_int
{     int * adval ;                    // adresse du tableau
                                       // des valeurs
      int nmax ;                       // nombre maxi d'éléments
      int nelem ;                      // nombre courant d'éléments
public :
      set_int (int = 20) ;             // constructeur
      set_int (set_int &) ;            // constructeur par recopie
      set_int & operator = (set_int &) ;  // opérateur d'affectation
      ~set_int () ;                    // destructeur
      int cardinal () ;                // cardinal de l'ensemble
      set_int & operator << (int) ;    // ajout d'un élément
      friend int operator % (int, set_int &) ;
                                       // appartenance d'un élément
          // envoi ensemble dans un flot
      friend ostream & operator << (ostream &, set_int &) ;
} ;
```

Voici ce que pourrait être la **définition** de notre classe (les points délicats sont commentés au sein même des instructions).

Définition de la classe *set_int*

```
#include "setint.h"
#include <iostream>
using namespace std ;
            /*************** constructeur ********************/
set_int::set_int (int dim)
{     adval = new int [nmax = dim] ;      // allocation tableau
                                          // de valeurs
      nelem = 0 ;
}
             /****************** destructeur ******************/
set_int::~set_int ()
{     delete adval ;                      // libération tableau
                                          // de valeurs
}
           /********** constructeur par recopie *************/
set_int::set_int (set_int & e)
{ adval = new int [nmax = e.nmax] ;// allocation nouveau tableau
nelem = e.nelem ;
int i ;
for (i=0 ; i<nelem ; i++)// copie ancien tableau dans nouveau
adval[i] = e.adval[i] ;
}
            /************ opérateur d'affectation ************/
set_int & set_int::operator = (set_int & e)   // commentaires faits
                                                 // pour   b = a
{     if (this != &e)                  // on ne fait rien pour a = a
        {  delete adval ;              // libération partie dynamique de b
           adval = new int [nmax = e.nmax] ;  // allocation nouvel
                                                 // ensemble pour a
           nelem = e.nelem ;           // dans lequel on recopie
           int i ;                     // entièrement l'ensemble b
           for (i=0 ; i<nelem ; i++)   // avec sa partie dynamique
             adval[i] = e.adval[i] ;
        }
      return * this ;
}
            /************ fonction membre cardinal ***********/
int set_int::cardinal ()
{     return nelem ;
}
             /************ opérateur d'ajout << ***************/
set_int & set_int::operator << (int nb)
{          // on examine si nb appartient déjà à l'ensemble
           //   en utilisant l'opérateur %
           // s'il n'y appartient pas, et s'il y a encore de la place
           //   on l'ajoute à l'ensemble
      if ( ! (nb % *this) && nelem < nmax )  adval [nelem++] = nb ;
      return (*this) ;
}
            /************ opérateur d'appartenance % **********/
int operator % (int nb, set_int & e)
{     int i=0 ;
                    // on examine si nb appartient déjà à l'ensemble
                    //  (dans ce cas i vaudra nele en fin de boucle)
while ( (i<e.nelem) && (e.adval[i] != nb) ) i++ ;
return (i<e.nelem) ;
}
             /****** opérateur << pour sortie sur un flot *****/
ostream & operator << (ostream & sortie, set_int & e)
{     sortie << "[ " ;
      int i ;
      for (i=0 ; i<e.nelem ; i++)
          sortie << e.adval[i] << " " ;
      sortie << "]" ;
      return sortie ;
}
```

Voici un **exemple de programme utilisant la classe *set_int***, accompagné du résultat fourni par son exécution.

Exemple : utilisation de la classe *set_int*

```
                /************* test de la classe set_int *********/
#include "setint.h"
#include <iostream>
using namespace std ;
main()
{     void fct (set_int) ;
      void fctref (set_int &) ;
      set_int ens ;
      cout << "donnez 10 entiers \n" ;
      int i, n ;
      for (i=0 ; i<10 ; i++)
          { cin >> n ;
       ens << n ;
          }
      cout << "il y a : " << ens.cardinal () << " entiers diff  rents\n" ;
      cout << "qui forment l\ ensemble : " << ens << "\n" ;
      fct (ens) ;
      cout << "au retour de fct, il y en a " << ens.cardinal () << "\n" ;
      cout << "qui forment l\ ensemble : " << ens << "\n" ;
      fctref (ens) ;
      cout << "au retour de fctref, il y en a " << ens.cardinal () << "\n" ;
      cout << "qui forment l\ ensemble : " << ens << "\n" ;
      cout << "appartenance de -1  : " <<  -1 % ens << "\n" ;
      cout << "appartenance de 500 : " << 500 % ens << "\n" ;
      set_int ensa, ensb ;
      ensa = ensb = ens ;
      cout << "ensemble a : " << ensa << "\n" ;
      cout << "ensemble b : " << ensb << "\n" ;
}

void fct (set_int e)
{     cout << "ensemble re  u par fct : " << e << "\n" ;
      e << -1 << -2 << -3 ;
}

void fctref (set_int & e)
{     cout << "ensemble re  u par fctref : " << e << "\n" ;
      e << -1 << -2 << -3 ;
}
```

```
donnez 10 entiers
3 5 3 1 8 5 1 7 7 3
il y a : 5 entiers diff  rents
qui forment l ensemble : [ 3 5 1 8 7 ]
ensemble re  u par fct : [ 3 5 1 8 7 ]
au retour de fct, il y en a 5
qui forment l ensemble : [ 3 5 1 8 7 ]
ensemble re  u par fctref : [ 3 5 1 8 7 ]
au retour de fctref, il y en a 8
qui forment l ensemble : [ 3 5 1 8 7 -1 -2 -3 ]
appartenance de -1  : 1
appartenance de 500 : 0
ensemble a : [ 3 5 1 8 7 -1 -2 -3 ]
ensemble b : [ 3 5 1 8 7 -1 -2 -3 ]
```

Remarque

Ici, il n'est pas possible d'agrandir l'ensemble au-delà de la limite qui lui a été impartie lors de sa construction. Il serait assez facile de remédier à cette lacune en modifiant sensiblement la fonction d'ajout d'un élément (*operator* <<). Il suffirait, en effet, qu'elle prévoit, lorsque la limite est atteinte, d'allouer un nouvel emplacement dynamique, par exemple d'une taille double de l'emplacement existant, d'y recopier l'actuel contenu et de libérer l'ancien emplacement (en actualisant convenablement les membres données de l'objet).

2. Exemple 2 : utilisation d'un patron de classes

Dans le précédent exemple, nous avons volontairement évité de recourir aux composants standards.

Or, si l'on ne s'intéresse qu'aux seules fonctionnalités de la classe, en dehors de la sortie sur un flot, celles-ci sont fournies intégralement par le **composant standard *set<int>*** (son utilisation nécessite l'inclusion du fichier en-tête *set*). En ce qui concerne la sortie sur un flot, on dispose de plusieurs démarches. On peut bien sûr la programmer au fur et à mesure des besoins, en écrivant à chaque fois les quelques instructions de parcours de l'ensemble :

```
set<int> ens ;
set<int>::iterator ie ;
.....
for (ie=ens.begin() ; ie!=ens.end() ; ie++)  cout << *ie << " " ;
```

On peut aussi en faire une fonction ordinaire, comme dans cet exemple, analogue à l'exemple d'utilisation précédent.

Premier exemple d'utilisation de la classe *set<int>*

```
#include <iostream>
#include <set>                          // pour set<int>
using namespace std ;
void af  che (set<int>) ;
main()
{     void fct (set<int>) ;
      void fctref (set<int> &) ;
      set<int> ens ;
      cout << "donnez 10 entiers \n" ;
      int i, n ;
      for (i=0 ; i<10 ; i++)
          { cin >> n ;
            ens.insert(n) ;
          }
      cout << "il y a : " << ens.size() << " entiers differents\n" ;
      cout << "qui forment l\ ensemble : " ; af  che(ens) ;
      fct (ens) ;
      cout << "au retour de fct, il y en a " << ens.size() << "\n" ;
      cout << "qui forment l\ ensemble : " ; af  che(ens) ;
      fctref (ens) ;
      cout << "au retour de fctref, il y en a " << ens.size() << "\n" ;
      cout << "qui forment l\ ensemble : " ; af  che(ens) ;
      cout << "appartenance de -1  : " << ens.count(-1)  << "\n" ;
      cout << "appartenance de 500 : " << ens.count(500) << "\n" ;
      set<int> ensa, ensb ;
      ensa = ensb = ens ;
      cout << "ensemble a : " ; af  che(ensa) ;
      cout << "ensemble b : " ; af  che(ensb) ;
}
void fct (set<int> e)
{     cout << "ensemble re  u par fct : " ; af  che(e) ;
      e.insert(-1) ; e.insert(-2) ; e.insert(-3) ;
}

void fctref (set<int> & e)
{     cout << "ensemble recu par fctref : " ; af  che(e) ;
      e.insert(-1) ; e.insert(-2) ; e.insert(-3) ;
}
void af  che (set<int> e)
      {set<int>::iterator ie ;
      cout << "[ " ;
      for (ie=e.begin() ; ie!=e.end() ; ie++)
          cout << *ie << " ";
      cout <<"] \n" ;
}
```

```
donnez 10 entiers
3 5 3 1 8 5 1 7 7 3
il y a : 5 entiers differents
qui forment l ensemble : [ 1 3 5 7 8 ]
ensemble recu par fct : [ 1 3 5 7 8 ]
au retour de fct, il y en a 5
qui forment l ensemble : [ 1 3 5 7 8 ]
ensemble recu par fctref : [ 1 3 5 7 8 ]
au retour de fctref, il y en a 8
qui forment l ensemble : [ -3 -2 -1 1 3 5 7 8 ]
appartenance de -1  : 1
appartenance de 500 : 0
ensemble a : [ -3 -2 -1 1 3 5 7 8 ]
ensemble b : [ -3 -2 -1 1 3 5 7 8 ]
```

On peut également créer artificiellement une **classe *set_int***, dérivée de *set<int>*, dans laquelle on surdéfinit l'opérateur <<, comme dans cet exemple qui fournit les mêmes résultats que le précédent.

Second exemple d'utilisation de la classe *set<int>*

```
#include <iostream>
#include <set>
using namespace std ;

          /************* d  claration de la classe set_int *********/
class set_int : public set<int>
{ public :
              // envoi ensemble dans un_ot
    friend ostream & operator << (ostream &, set_int &) ;
} ;

           /************* d     nition de la classe set_int *********/
ostream & operator << (ostream & sortie, set_int & e)
{    sortie << "[ " ;
     set<int>::iterator ie ;
     for (ie=e.begin() ; ie!=e.end() ; ie++)
        sortie << *ie << " " ;
     sortie << "]" ;
     return sortie ;
}

              /************* test de la classe set_int *********/
main()
{
     void fct (set_int) ;
     void fctref (set_int &) ;
     set_int ens ;
     cout << "donnez 10 entiers \n" ;
     int i, n ;
     for (i=0 ; i<10 ; i++)
        { cin >> n ;
          ens.insert(n) ;
        }
     cout << "il y a : " << ens.size() << " entiers differents\n" ;
     cout << "qui forment l\ensemble : " << ens << "\n" ;
     fct (ens) ;
     cout << "au retour de fct, il y en a " << ens.size() << "\n" ;
     cout << "qui forment l\ensemble : " << ens << "\n" ;
     fctref (ens) ;
     cout << "au retour de fctref, il y en a " << ens.size() << "\n" ;
     cout << "qui forment l\ensemble : " << ens << "\n" ;
     cout << "appartenance de -1  : " << ens.count(-1)  << "\n" ;
     cout << "appartenance de 500 : " << ens.count(500) << "\n" ;
     set_int ensa, ensb ;
     ensa = ensb = ens ;
     cout << "ensemble a : " << ensa << "\n" ;
     cout << "ensemble b : " << ensb << "\n" ;
}
void fct (set_int e)
{    cout << "ensemble re  u par fct : " << e << "\n" ;
     e.insert(-1) ; e.insert(-2) ; e.insert(-3) ;
}
void fctref (set_int & e)
{    cout << "ensemble re  u par fctref : " << e << "\n" ;
     e.insert(-1) ; e.insert(-2) ; e.insert(-3) ;
}
```

Remarque

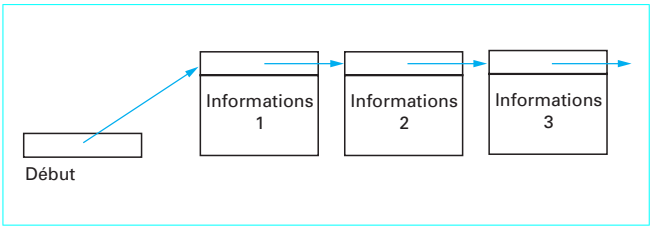
Ici, il n'est pas nécessaire de redéfinir l'affectation ou le constructeur par recopie. En effet, compte tenu des règles relatives à l'héritage, les fonctions par défaut appellent bien les fonctions voulues dans la classe de base ; cela suffit ici puisque la classe dérivée n'introduit aucune partie dynamique supplémentaire.

3. Exemple 3 : liste hétérogène

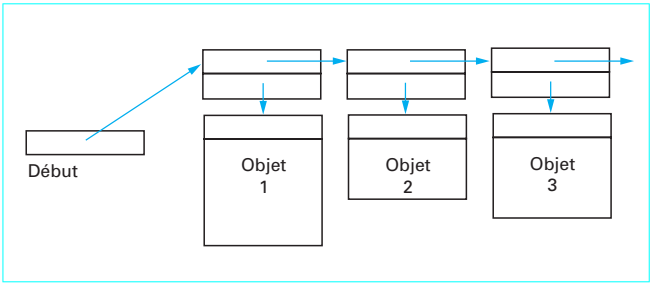
Nous allons créer une **classe permettant de gérer une liste chaînée d'objets de types différents** et disposant des fonctionnalités suivantes :

— ajout d'un nouvel élément ;
— affichage des valeurs de tous les éléments de la liste ;
— mécanisme de parcours de la liste.

Rappelons que, dans une liste chaînée, chaque élément comporte un pointeur sur l'élément suivant. En outre, un pointeur désigne le premier élément de la liste. Cela correspond à ce schéma :



Mais ici l'on souhaite que les différentes informations puissent être de types différents. Aussi chercherons nous à isoler dans une classe (nommée *liste*) toutes les fonctionnalités de gestion de la liste elle-même sans entrer dans les détails spécifiques aux objets concernés. Nous appliquerons alors ce schéma :



La classe *liste* elle-même se contentera donc de gérer des éléments simples réduits chacun à :

— un pointeur sur l'élément suivant ;
— un pointeur sur l'information associée (en fait, ici, un objet).

On voit donc que la classe va posséder au moins :

— un membre donnée : pointeur sur le premier élément (*debut*, dans notre schéma) ;
— une fonction membre destinée à insérer dans la liste un objet dont on lui fournira l'adresse (nous choisirons l'insertion en début de liste, par souci de simplification).

L'affichage des éléments de la liste se fera en appelant une méthode *affiche*, spécifique à l'objet concerné. Cela implique la mise en oeuvre de la ligature dynamique par le biais des fonctions virtuelles. La fonction *affiche* sera définie dans un premier type d'objet (nommé ici *mere*) et redéfinie dans chacune de ses descendantes.

En définitive, on pourra gérer une liste d'objets de types différents sous réserve que les classes correspondantes soient toutes dérivées d'une même classe de base. Cela peut sembler quelque peu restrictif. En fait, cette « famille de classes » peut toujours être obtenue par la création d'une classe abstraite (réduite au minimum, éventuellement à une fonction *affiche* vide ou virtuelle pure) destinée simplement à donner naissance aux classes concernées. Bien entendu, cela n'est concevable que si les classes en question ne sont pas déjà figées (car il faut qu'elles héritent de cette classe abstraite).

D'où une **première ébauche de la classe *liste*** :

```
struct element                  // structure d un   l  ment de liste
{    element * suivant ;        // pointeur sur l   l  ment suivant
     mere * contenu ;           // pointeur sur un objet quelconque
} ;
class liste
{    element * debut ;          // pointeur sur premier   l  ment
public :
     liste () ;                 // constructeur
     ~liste () ;                // destructeur
     void ajoute (mere *) ;     // ajoute un   l  ment en d  but de liste
     void af  che () ;
     .....
} ;
```

Pour mettre en œuvre le parcours de la liste, nous prévoyons des fonctions élémentaires pour :

— initialiser le parcours ;
— avancer d'un élément.

Celles-ci nécessitent un « **pointeur sur un élément courant** ». Il sera membre donnée de notre classe *liste* ; nous le nommerons *courant*. Par ailleurs, les deux fonctions membres évoquées doivent fournir en retour une information concernant l'objet courant. À ce niveau, on peut choisir entre :

— l'adresse de l'élément courant ;
— l'adresse de l'objet courant (c'est-à-dire l'objet pointé par l'élément courant) ;
— la valeur de l'élément courant.

La deuxième solution semble la plus naturelle. Il faut simplement fournir à l'utilisateur un moyen de détecter la fin de liste. Nous prévoirons donc une fonction supplémentaire permettant de savoir si la fin de liste est atteinte (en toute rigueur, nous aurions aussi pu fournir un pointeur nul comme adresse de l'objet courant ; mais ce serait moins pratique car il faudrait obligatoirement agir sur le pointeur de liste avant de savoir si l'on est à la fin).

En définitive, nous introduisons **trois nouvelles fonctions membres** :

```
void * premier () ;
void * prochain () ;
int   ni () ;
```

Comme dans le premier exemple, il nous faut nous préoccuper des **problèmes d'affectation et de copie d'objets de type *liste***. Ici, nous nous contenterons d'interdire ces opérations à l'utilisateur en levant une exception lorsque l'une d'entre elles sera tentée. Pour ce faire, nous créons (un peu artificiellement) deux classes *exc_affec* et *exc_copie* (en fait vides). Pour faciliter la gestion des exceptions correspondantes, nous les faisons toutes deux dériver d'une troisième nommée ***exc_liste***.

Voici la **liste complète des différentes classes voulues**. Notez qu'ici, nous avons regroupé déclaration et définition de classes. De plus, au sein des déclarations, nous avons exploité la possibilité qu'offre C++ de définir directement certaines fonctions que l'on qualifie alors de « fonctions en ligne ». Cette façon de procéder simplifie les définitions courtes. En outre, elle offre au compilateur la possibilité (pas l'obligation) d'introduire le code objet correspondant à chaque appel, en évitant les « changements de contexte » qui seraient induits par un appel classique (sauvegarde de registres, conservation d'une adresse de retour...).

Enfin, nous avons réalisé un petit **programme d'essai** de la classe liste, en définissant deux classes *point* et *complexe* (lesquelles n'ont pas besoin de dériver l'une de l'autre), dérivées de la classe abstraite *mere* et dotées chacune d'une fonction *affiche* appropriée. Nous y avons tenté une affectation entre objets de type *liste* et nous interceptons convenablement l'exception *exc_liste*.

Déclaration, définition et utilisation d'une liste hétérogène

```
#include <iostream>
#include <cstddef>                          // pour la définition de NULL
using namespace std ;
                        // ************* classes exceptions *************
class exc_liste {} ;
class exc_affec : public exc_liste {} ;
class exc_copie : public exc_liste {} ;
                                // ************* classe mere *************
class mere
{       public :
            virtual void affiche () = 0 ;       // fonction virtuelle pure
} ;
                                // ************* classe liste *************
struct element                              // structure d'un élément de liste
{       element * suivant ;                 // pointeur sur l'élément suivant
        mere * contenu ;                    // pointeur sur un objet quelconque
} ;
class liste
{       element * debut ;                   // pointeur sur premier élément
        element * courant ;                 // pointeur sur élément courant
public :
    liste ()                                // constructeur
            { debut = NULL ; courant = debut ; }
    ~liste () ;                             // destructeur
    void ajoute (mere *) ;                  // ajoute un élément
    void premier ()                         // positionne sur premier élément
            { courant = debut ;
            }
    mere * prochain ()                      // fournit l'adresse de l'élément
                                            // courant (0 si fin)
                                            // et positionne sur prochain
                                            // élément (rien si fin)
    {mere * adsuiv = NULL ;
            if (courant != NULL) {  adsuiv = courant -> contenu ;
                                    courant = courant -> suivant ;
                                 }
            return adsuiv ;
    }
    void affiche_liste () ;                 // affiche tous les éléments
                                            // de la liste
    int fini () { return (courant == NULL) ; }
    liste & operator = (liste & l) { throw exc_affec() ; }
    liste (liste &) { throw exc_copie() ; }
} ;
liste::~liste ()
{   element * suiv ;
    courant = debut ;
    while (courant != NULL )
      { suiv = courant->suivant ; delete courant ; courant = suiv ; }
}
void liste::ajoute (mere * chose)
{   element * adel = new element ;
    adel->suivant = debut ;
    adel->contenu = chose ;
    debut = adel ;
}
void liste::affiche_liste ()
{   mere * ptr ;
    premier() ;
    while ( ! fini() )
      { ptr = (mere *) prochain() ;
        ptr->affiche () ;
      }
}
                                // ************* classe point *************
class point : public mere
{       int x, y ;
    public :
      point (int abs=0, int ord=0) { x=abs ; y=ord ; }
      void affiche ()
          { cout << "Point de coordonnees : " << x << " " << y << "\n" ; }
} ;
                                // ************* classe complexe *************
class complexe ; public mere
{       double reel, imag ;
    public :
      complexe (double r=0, double i=0) { reel=r ; imag=i ; }
      void affiche ()
          { cout << "Complexe : " << reel << " + " << imag << "i\n" ; }
} ;
                                // ************* programme d'essai *************
main()
{       try
        {   liste l1 ;
            point a(2,3), b(5,9) ;
            complexe x(4.5,2.7), y(2.35,4.86) ;
            l1.ajoute (&a) ; l1.ajoute (&x) ; l1.affiche_liste () ;
            cout << "--------\n" ;
            l1.ajoute (&y) ; l1.ajoute (&b) ; l1.affiche_liste () ;
            liste l2 ;
            l2 = l1 ;  // provoque une exception exc_affec ;
        }
        catch (exc_liste)
        {   cout << "tentative de copie ou d'affectation de liste" ;
        }
}
```

```
Complexe : 4.5 + 2.7i
Point de coordonnees : 2 3
--------
Point de coordonnees : 5 9
Complexe : 2.35 + 4.86i
Complexe : 4.5 + 2.7i
Point de coordonnees : 2 3
tentative de copie ou affectation de liste
```